Name: Ayush Bothra

Branch: Computer engineering

Roll No: 231070011

DAA Assignment 2

Aim: To create algorithms for linear and binary search and code them according to the coding style of

the Language used.

Algorithm:

linear\_search(array, target):

// Input: The function takes an array that has a list of elements and the target – the element to be

found in the array – as its inputs

// Output: The function then outputs either the index of the target – if it is found in the array – or

returns “-1”, if the element was not found in the array.

for i <- 0 to length(array):

if array[i] == target do:

return i

end for

return -1

binary\_search(array, target, left, right):

// Input: The function takes an array that has a list of elements, a target – the element to be found in

the array, left – the left of the array – and the right – the right of the array, as inputs.

// Output: The function outputs either the index of the target if it is found in the array, or returns -1 if the target does not exist in the array.

if left >= right do:

return -1

middle <- left + (right - left) // 2

if array[middle] == target do:

return middle

if array[middle] < target do:

return binary\_search(array, target, middle + 1, right)

if array[middle] > target do:

return binary\_search(array, target, left, middle – 1)

Code:

"""

Name: Ayush Bothra

Roll No: 231070011

Date: 10-08-2024

Algorithm: Linear and Binary Search

"""

def linear\_search(a, target):

    """Function to search an array by linear search."""

    for idx, value in enumerate(a):

        if value == target:

            return idx

    return -1

def sorted(array):

    for i in range(len(array)-1):

        # Checking if the array is sorted

        if array[i] > array[i + 1]:

            return False

    return True

def binary\_search(a, target, left, right):

    """Recursive function to search an array by binary search."""

    if left > right:

        return -1

    middle = left + (right - left) // 2

    if a[middle] == target:

        return middle

    if a[middle] < target:

        return binary\_search(a, target, middle + 1, right)

    if a[middle] > target:

        return binary\_search(a, target, left, middle - 1)

if \_\_name\_\_ == "\_\_main\_\_":

    # using a dictionary to store multiple test-cases.

    # the key is target, the value is the array.

    linear\_tests = {

        8: [3, 7, 4, 1, 2, 6],

        9: [],

        12: [-12, 13, 8, 16, 12],

        200: [7, -30, -100, 200, 8],

        -1: [2, 7, 9, -10, -1, -3],

        }

    binary\_tests = {

        3: [3, 6, 7, 2, 1, -1],

        8: [-1, 1, 2, 4, 8],

        -2: [-1, 2, 3, 6, 8, 10],

        4: [-10, -5, 4, 7],

        9: [2, 7, 9, 11, 13],

        }

    print("output for linear search:")

    for key  in linear\_tests.keys():

        print(f"array: {linear\_tests[key]}, target: {key}")

        print(f"element position: {linear\_search(linear\_tests[key],key)}")

    print("# ------------------------------------------------ #")

    print("output for binary search: ")

    for key in binary\_tests.keys():

        print(f"array: {binary\_tests[key]}, target: {key}")

        if sorted(binary\_tests[key]):

            position = binary\_search(binary\_tests[key],key,

                                     0,len(binary\_tests[key]))

            print(f"element position: {position}")

        else:

            print("element position: -1")

Test-Cases:

linear\_tests = {

        8: [3, 7, 4, 1, 2, 6],

        9: [],

        12: [-12, 13, 8, 16, 12],

        200: [7, -30, -100, 200, 8],

        -1: [2, 7, 9, -10, -1, -3],

        }

Expected output:

1. The element does not exist, hence -1 is the output
2. The array is empty, hence -1 is the output
3. The element will be found at the index 4, hence 4 is the output
4. The element will be found at index 3, hence 3 is the output
5. The element will be found at index 3, hence 4 is the output

    binary\_tests = {

        3: [3, 6, 7, 2, 1, -1],

        8: [-1, 1, 2, 4, 8],

        -2: [-1, 2, 3, 6, 8, 10],

        4: [-10, -5, 4, 7],

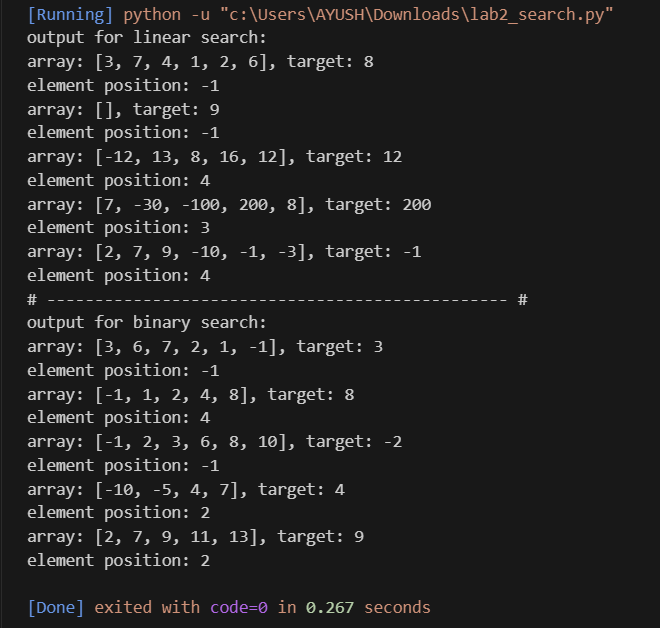
        9: [2, 7, 9, 11, 13],

        }

Expected Outputs:

1. The array is not sorted, hence -1 is the output
2. The element will be found at index 4, hence 4 is the output
3. The element is not present in the array, hence -1 is the output.
4. The element will be found at index 2, hence 2 is the output
5. The element will be found at index 2, hence 2 is the output

Output:



Time Complexity of the two algorithms:

For linear search: The smallest operation is the comparison of target with each element. Let this operation take 1 unit of time.

Then, total time will be summation of this operation from i = 0 to n – 1, where n is the size of the array. This will then simplify to give T(n) = n + 1, where 1 is added for the time taken for the return statement.

Thus, the asymptotic Time Complexity is O(n)

For binary search: This time, due to recursion, the total size of the array gets divided into half and then that half is used for calculations. This is why we can say: T(n) = T(n / 2) + 1, where the one is for the return statement.

Let n = 2^i for i elements, then, T(2^i) = T(2^i – 1) + 1

= T(2^i-2) + 2

= T(1) + I where T(1) = 1.

Therefore, we can now write this as: T(n) = logn + 1. Hence, the Time Complexity of this search is O(logn).

Conclusions:

We created an algorithm for linear search:

* This search uses a single pointer to traverse through-out the given array and compares each value present in the array with the target, if at any point this comparison returns a true value, it will return the index of the element in the array.
* This is a time-consuming search and is not recommended for very large arrays since it has a time complexity of O(n)

We also created an algorithm for binary search:

* This search uses two pointers: the left of the array and the right of the array. These pointers are use to calculate the position of the middle of the array. Then the element in the middle is compared with the target and the positions of left and right pointers are changed accordingly. If the comparison returns a true value, the algorithm will return the middle.
* This is a much faster search than linear search and has a time complexity of O(logn) but it needs a sorted array to work. This is the only condition for binary search. It is a form of divide and conquer algorithm.

We also read about coding styles:

* Knowing a coding style and sticking to it is important as it allows a generalization of code writing methods due to which others can also understand the code written by on person better than if it would have been written without any rules in mind.